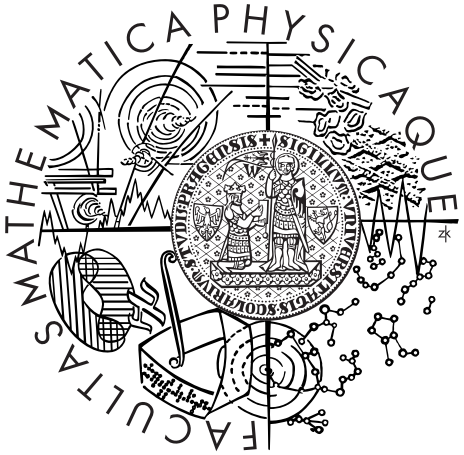
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Cyril Hrubiš

Knihovna pro tvorbu uživatelského rozhraní

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: Mgr. Ondřej Zajíček

Study programme: Computer Science (IOI)

Prague 2012

This thesis is dedicated to various people without whom these pages wouldn't be the same or even exist at all. In short, to the authors of UNIX and C programming language; to Richard Stallman who started the Free Software Foundation and to Linus Torvalds who began the Linux kernel project.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Knihovna pro tvorbu uživatelského rozhraní

Autor: Cyril Hrubiš

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Mgr. Ondřej Zajíček, Katedra teoretické informatiky a matematické logiky

Abstrakt:

Tato práce se zabývá rozborem a implementací knihovny uživatelského rozhraní pro operační systém Linux s důrazem na malá zařízení.

Knihovna je naimplementována v jazyce C pro operační systém Linux a sama se skládá z rozhraní pro psaní aplikací a backendu což je část, která zobrazuje uživatelské rozhraní samotné. Aplikace a backend běží jako samotné procesy, které komunikují přes sockety a navíc je mezi nimi multipexor, který jednak izoluje aplikace od backendu a zároveň umožnuje připojení více backendů najednou. Backend je koncipován jako prostředí pro malé zařízení například PDA či tablety.

Klíčová slova: Uživatelské rozhraní, Widget toolkit, Linux

Title: User Interface library

Author: Cyril Hrubiš

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Ondřej Zajíček, Department of Theoretical Computer Science and Mathematical Logic

Abstract: This work is dedicated to theoretical discussion and implementation of user interface library for Linux targeted primarily to small devices.

The library is implemented for Linux operating system using the C programming language and is divided into an interface for writing applications and backend which is the part that renders the user interface. Applications and backend are running as separate processes and communicate over sockets, moreover there is a multiplexer inbetween which both isolates applications from backend and allows for multiple backends to be connected at the same time. Backend is designed for small devices such as PDAs or tablets.

Keywords: User Interface, Widget toolkit, Linux

Table of Contents

1. Motivation

1.1 Preface

Computers are, without a discussion, common part of our life, they exist in many forms and it's very unlikely that they would vanish in the future. But despite their complexity, computers are just machines created to perform tasks. We could spend an eternity trying to precisely define what a task is however, for the purpose of this document, an intuitive and lousy definition is enough. A task is what people are able and what they like to do using a computer. It may be looking for the nearest public transport station, sending messages to friends, playing music, using computer for solving equations or adjusting the temperature in a home heating system.

To perform a task, a machine needs to be programmed, since the times computers were hardwired for a single task are long gone. The program (application) is a software installed on a computer. One could think of the software as of a machine created to perform a particular task. In order to operate the machine there must be a UI (User Interface) of some kind. Continuing the parallel with an old-fashioned machine the UI is its front panel.

It would be naive to think that UI was invented from a scratch when computer industry was born. The basic ideas, as said before, may be linked to the old-fashioned machines. The textual interfaces are using writing system and symbols that existed basically since people invented writing. Let's take a closer look at several types of the human-computer interactions to see how they looks like and how they involved over the time.

1.2 Static configuration UI

Static configuration is the simplest type of UI. One could argue that it's not a UI at all as a user is not able to change the configuration at run-time or interact with the running machine in any way. On the other hand some parts of the theoretical discussions may apply even for such simple UI.

The static configuration is a configuration stored in a well defined format. It may be raw hexadecimal data loaded into flash memory chip of a digital terrestrial receiver or of a car radio as well as a UNIX [1] daemon configuration written in a plain text file. In a GUI it's included implicitly as default widget values (initial positions of knobs on the front panel).

The UI interaction in this case is simply loading an updated firmware/config or editing a configuration file and restarting the application (turning the machine off, changing settings and turning it back on).

The notable fact is that even this UI has to be structured somehow and this is done by the 'well defined format'.

For the case of hexadecimal data, every value is (usually) located at a specific offset and has predefined type. The offset simply says position in the data where

[1] UNIX Operating System `http://www.unix.org/`

the value starts and the type defines length and information about the way the value should be interpreted.

In the case of daemon configuration the file format usually consists of `key = value` pairs written in human-readable form but the idea is still the same. The values must be identified somehow and their format must be defined.

1.3 Command line UI

Command line UI or CLI (Command Line Interface) is probably the oldest kind of the interactive human-computer interface. The computer is operated using a textual form by writing commands into an interpreter and reading its output. This kind of UI has roots in written communication and despite the fact that the interpreter is much stricter than a human reader (the interpreter takes everything literally) I find this type of human-computer interaction quite natural. Another notable fact is that this kind of UI is also conceptually close to programming.

Despite its age the CLI is still useful especially for certain tasks. It is fairly easy to transfer the commands and their output over the network and thus operate several machines across the world from a single location.

Another powerful feature of CLI is scripting. The interpreter is usually able to run scripts - sequences of commands written into a file. Which allows for efficient automation of daily tasks.

The CLI is however not so useful for heavily interactive applications. Consider for an example a mail client; although it's possible to have a mail client operated by CLI (see mailx[2]), I personally prefer mail that shows data in a more structured manner and displays changes in the mailbox interactively. Do not confuse a TUI (text user interface which renders a graphical application using text characters in the terminal) with CLI described here. There are, in fact, several mail clients written in TUI that are quite usable. The TUI is, in terms of functionality, more closer to GUI discussed in the next paragraph.

1.4 Graphical UI

In the old times, before the computer age, machine controls were hardwired into the front panel. That is still common for most of the daily used machines (image an old fashioned radio or tape recorder), however the number of computer operated machines seems to increase. The GUI (Graphical User Interface) is inspired by the traditional machine controls in a sense that GUI renders controls that may look similar to these used on front panels.

Despite the fact that the computer GUI has evolved over the time the basic ideas are still the same. The application (machine) has some number of windows (front panels) each of them consists of widgets (buttons, knobs, switches, light bulbs etc.). The information about the task state is propagated to the user through these controls and, at the same time, runtime parameters are adjusted, using these controls. The main difference between a computer and an old fashioned machine is that user controls could be created and destroyed at a run-time.

[2] `http://pubs.opengroup.org/onlinepubs/009604499/utilities/mailx.html`

The GUI is often called intuitive because of the connection to the real-life machines and since its creation it's widely accepted. As a matter of fact, all of widely used desktop operating systems have (sometimes more than one) complex GUI toolkits. There are even usability experts who study the structure and intuitiveness of the GUI.

However useful the GUI is, there are also major drawbacks. The GUI is designed for people who are not computer experts or have very little computer knowledge. In order to help them to operate the machine, GUI must hide some amount of the details. In order to do so and in order to explain the functionality to common users it tries to create parallels to real life or to some common knowledge. The problem is that sometimes the result is too simplified or the parallel too vague to be useful. In such circumstances the result will confuse both computer expert and the common user. The problem with such approach is that it's not easy to balance right amount of abstraction and parallels.

1.5 Widget Toolkit

Widget toolkit is a software library with a predefined set of ready-to-use widgets. Its purpose is to ease GUI creation so the programmer can concentrate on application code that does the actual job instead of spending majority of the time on a code that draws "buttons" on the screen.

Widget toolkit API (Application Programming Interface) usually provides relatively easy way to create and manage application windows and widgets.

The purpose of the Micro Witchcraft library is to create reasonable and usable API as well as to try to include some of the features of other UIs.

2. Widgets

2.1 Widget as an abstract value

The widget, as an abstract object, is a visual representation of data. The data the widget represents may vary ranging from single number to a chosen subset of a set of values or a tree representing recursive directory structure.

From the user's point of view the value may or may not be writable (you wouldn't want to be able to change labels on machine front panel by accident, would you?). Sometimes it can be even meaningful to define a value that is only writable so nobody can see it. The most obvious example are various password forms.

From a different point of view, widget values are conceptually similar to a system of types in a programming language. There are numeric widgets whose abstract value corresponds to sliders, spin buttons, edit boxes for numbers, progress bars, etc. All these widgets are just numbers with minimum and maximum, some are and some aren't modifiable from the UI. Then there are obvious cases for widgets whose values are strings. This group of widgets includes labels, various input text boxes, etc. To describe more complex types (for example various tables, lists and sets) we need to group several values into one entity. Speaking in terms of C programming language we speak of structures and arrays. Take, as an example, several select boxes that represent choices from a set of values (usually strings describing the possible values). This widget could be defined as a choice from a set which, translated to the terms of programming language, could be an index into an array of strings.

Now let's look closer on some of the commonly used widgets.

Label Widget is a string that is not writable from the UI. Its value is a string that is simply shown somewhere in the interface. When using the front panel parallel it's either a label printed on the panel surface or a display that could show a line of text.

Slider Widget is a signed or unsigned integer number bounded by maximum and minimum. This widget is writable from the UI. Its value is represented by a position of an indicator on a line representing the range. The value is changed by dragging the indicator from one end to another. This kind of interface is commonly used on audio mixing pults.

Progress Bar is unsigned integer value that is not writeable from UI. Progress bar is drawn as a bar that grows as the progress continues. Additionally it may show some information about the nature of the progress, i.e. number of seconds from the start of a song, size of the file that has been downloaded etc.

Spin Button Widget abstract value is very same as for the Slider Widget however the visual representation is very different. Spin button is usually repre-

sented as a numeric value along with two arrow-labeled buttons that increase or decrease its value.

List of Select Boxes is a structure that contains list of labels from which exactly one could be selected at the time. The visual representation here is a group of labels where each of them has small circle and for the circle for the selected choice is filled with different color. The same interface may be shown as a menu in which the the selected item is highlighted by different background color or possibly by a spin button.

These examples as well as theoretical discussion are trying to suggest that the widget value and its (possibly graphical) representation are orthogonal concepts. The conclusion is that library API should rather concentrate on widget value and behavior rather than how it's presented to the user.

The real problem however is where to draw a line, how abstract the widget toolkit API should be? Which is unfortunately not an easy question to answer. And after great deal of research I've decided to follow my instincts and tried to create easy to use API rather than possibly cleaner abstract system which may be harder work with.

2.2 Widget in terms of reactive programming

Reactive programming is a paradigm that is oriented around data flows and propagation of a change. Basically this concept suggests that a programmer should focus more on what the program does rather than how to express it in the terms of a particular programming language.

As stated before, widget is a control. To control a task, change in the widget value must be able to trigger an action and possibly propagate into other widgets. For example pressing "Next" button in music player will affect most of the widgets that represents the player UI (or at least change the current track position and name).

In the terms of C programming language this means callback. Callback is a pointer to a function that may be called. So most of the widgets have a callback to be called when user changes a widget value from within a UI.

2.3 Widgets as graphical representation of data

So far we have been looking at widgets as abstract values or as triggers for an action. Now we will focus on another, probably the most complex topic, which is widget graphical representation.

The graphical representation of a widget is not an easy task. As stated before there are even usability engineering studies and experts. This broad field of study includes psychology, engineering, industrial design, graphic design and more. Concentrating all these knowledge to bring consistent, intuitive and effective UI.

Focusing more on the technical side of the problem we start with a window that consists of smaller units called widgets. So one problem to solve is the layout

of widgets in the window. The result should be well structured and generally pleasing to the eye. As the graphical representation of controls consists from text and symbols, a window may be considered either as a page of a text or machine front panel. Now, 'pleasing to the eye' can be better defined through either typography or design. Both of them however still have some amount of applied art included.

Another problem to solve is dynamic nature of the GUI. Various font faces tend to have very different glyph metrics[1] and so interface designed with particular font in mind would certainly break after a font change.

Also different screens have different DPI[2] thus GUI metrics shouldn't count sizes in terms of pixels but rather some other metrics. I've decided to derived size measuring unit from currently selected font. Speaking in terms of typography the basic unit of measure is 1em. Moreover devices range greatly in screen pixel resolution. The GUI must be able to manage and distribute available space (which is sometimes called white space) and must be able to shrink and grow as needed. When most of the sizes are based on a font size, part of the problem is solved by simply choosing adequate font size.

[1]Table of sizes of elements of a font, for example glyph metric of letter A is a width and height of the letter as well as information about it's positioning regarding the to the surrounding glyphs in a text.

[2]Dots per inch; a measure of physical pixel density.

3. Implementation

3.1 Overall design

The Micro Witchcraft widget toolkit is split into three parts that communicate over sockets.

- Applications
- Multiplexer
- Drawing backends

3.1.1 Applications

Application is a program that does the actual work, for example it plays the music, loads and renders documents, etc. Each application runs as a separate UNIX process and is connected to the multiplexer (MW_Appd).

3.1.2 Multiplexer

The application daemon (MW_Appd) is a multiplexer that behaves as a proxy between applications and render backends. The advantages of this such design are described in the next paragraphs.

3.1.3 Drawing backends

The last group, as seen by functionality, are rendering backends, where the actual GUI or UI is rendered or exposed in some way and presented to the user.

3.1.4 Design choices explained

First of all the most obvious decision: The applications needs to be isolated in order not to interfere with others so that bug in one application cannot easily cause failure of another one. This requirement intuitively leads to one application per process design where application memory and other resources are isolated by the operating system (which is common design pattern in most cases, although there are systems that don't have/support MMU[1] such as DOS[2] or uClinux[3] where such isolation is not possible). Also note that not all of the application resources are isolated, for an example the filesystem structure is shared between all applications which could still lead to failures and crashes.

The needs for application multiplexer are a little different. The main motivation for this arrangement is that bug in the rendering backend (such as dereferencing wrong pointer) will not (most likely) affect running applications. You may

[1] Memory Management Unit - hardware component that handles virtual memory and protection

[2] Disk Operating System `http://en.wikipedia.org/wiki/DOS`

[3] `http://www.uclinux.org/`

argue that there is a possibility for a bug in the multiplexer code too, which would still cause all applications to crash, however the amount of multiplexer code is quite small in comparison to the rendering backend, so it's far easier to review and fix its code. Moreover in such setup, the graphics backend could be restarted without closing all applications or transparently moved to another machine or even several (possibly different) backends could be running at the same time. The multiplexer could be also used for "migrating" the GUI from one machine to another. The drawback is a latency in the communication as the data needs to travel to the multiplexer first.

3.2 The Application and Backend split

Another problem to decide, once we split the widget toolkit into an application and rendering backend is where and how the drawing is done. Traditionally on a UNIX workstation drawing is done mostly by the application in a sense that applications decides where the lines are drawn, application loads images and puts them into a window. The actual drawing may be and is done by an X server, OpenGL[4] or anything else. The important fact is that application decides what and where is rendered. On the other hand there are widget toolkits that provide more abstract interface for creating user interfaces. Once you decide to use widget toolkit such as GTK[5] or Qt[6], you are no longer using the X server API to create the user interface by drawing rectangles and text. Rather than that you use the widget toolkit API that draws the widgets for you and, as the matter of fact, you are no longer in precise control of what is being drawn and how (at least for most of the time).

This naturally leads to a solution that moves the rendering part of the widget toolkit out of the application making it smaller and simpler. So rather than letting the application to draw its GUI, the abstract data types and rendering hints are send by the application. Not surprisingly similar solutions do already exists and are in use in various applications.

The concept could roughly be compared to the Web and CSS styles which is widely used today. Web pages are written in HTML and CSS languages which describe both form and content. The Web pages are sent to the Web browser over the network and the Web browser renders the GUI or some other UI (for example text to speech synthesizers that allows blind people to 'read' the Web pages too).

As a less obvious example there are xmms2[7] and mpd[8] both programs are used for playing music. Both of them have a well defined interface to describe music player state and song library. The actual user interface is rendered by a program that connects to the daemon. The daemon does the job of playing music and allows to be controlled from several computers over the network. In these cases, however, no or nearly none information about the actual look of the GUI is

[4] `http://www.opengl.org/`
[5] `http://www.gtk.org/`
[6] `http://qt-project.org/`
[7] XMMS2 Music Player `http://xmms2.org/`
[8] Music Player Daemon `http://mpd.wikia.com/`

exported (by the daemon player itself). In these cases the information is defined more or less explicitly by the fact that the protocol describes a music player.

Surprisingly there are similar concepts build in the Linux kernel and system libraries. For example the ALSA[9] mixer API allows you to control the sound volume and other sound related settings from several programs at once. These changes must be propagated between the mixers in order to stay synchronized and are broadcasted by the ALSA library to all currently running sound mixers. As in the previous example ALSA mixer API doesn't export any information about how the control widgets look or are placed in the application which is again possible because everybody who uses computer frequently knows, at least roughly, how sound mixer interface should look. In general case this is not an option and so widget toolkit needs to carry at least minimal information about the placement of widgets on the screen.

Such approach where widget structure is exported in well defined manner also gives us some degree of freedom how to present the application interface to the user and how to interpret user input without need to rewrite all existing applications or the widget library. All this would take is to tweak the backend renderer. This is especially useful on small devices and gadgets with unusual screen and human input interfaces. We could even write different user interfaces for existing applications without the need for rewriting application itself. Take as an example music player, now if you have particular buttons on your device (possibly a small music player with display and a few buttons) and you decide to use them for controlling what song is playing, all that needs to be done is backend that sends button pushes to the right application, which could be done with a few lines of code. Also, if the way application exports it's interface is done right, the widget toolkit could be fully scriptable, so controlling music player could be just a matter of writing a short script. Scriptable widget toolkit is also very useful for automated testing which greatly eases writing of automated tests.

3.3 Serialization and Communication Protocol

Once we decided on what is done by an application and what is done by the renderer backend the next step is to design an interface that would allow us pass the data from application to backend and back. And as the application is, for a good reason, isolated from the backend by running in separate process, we need a way to serialize the data in order to exchange them between applications and backends.

3.3.1 Application part

The serialization, from an application point of view, composes of several problems to solve.

First we need to be able to identify objects whose values we want to get or set. In other words, we need to resolve a given identifier either to a value, or to an address in memory.

[9] Advanced Linux Sound Architecture `http://www.alsa-project.org`

To solve this problem I've chosen to create recursive namespace that would globally identify each exported object in application. The solution is similar to the filesystem structure, where application is filesystem root, window is a directory and widget attributes are files (possibly grouped in directories). In contrast with filesystem implementations, so called files here have types that more or less corresponds to types defined in the C programming language. The advantages of such arrangement are scriptability (once widget and its attributes have well defined and fixed path in such structure you can easily press application buttons from a script). The textual format of the interface was chosen in order to ease debugging as the debug messages are easier to read and interpret.

In order not to be confused with a filesystem paths a colon, rather than slash, is used as a separator character. Example path, in running application, may then look like 'win:button:state' where state may be readable and writeable boolean value. Writing true into this value may press the button.

Once we could identify each application object (mostly widget attributes) whose value needs to be read or written from backend we need a way to set or get it's value. Moreover, in case of widgets, some of the value changes need to trigger an action (imagine a button was pressed and we need to call a button callback and broadcast to all backends that button has been pressed). So just changing a value stored in a memory wouldn't be sufficient as in such case the button would be pressed 'silently'. In some cases the value may be set only to certain subset of values (slider value should be set only within the bounds) so we need to make sure nobody is able to set it otherwise. To satisfy these conditions widget attribute values are read (get) or modified (set) only from appropriate callback functions.

Some of the values may be readonly from a backend view and in such case the set function is not defined. Imagine, for an example, slider bounds these could be changed from within the application, but changing them from withing a render is surely wrong thing to do.

Now let's look at the programming language types. I've chosen to use the C programming language so the set/get functions needs to work on C types. The types in C consists of scalar values (eg. integer, float, char, ...) and allows us to construct (recursively) more complicated data structures by using arrays and structures. The structures maps nicely to the 'directories' in our recursive namespace. The arrays needs special treatment though and the rest of chosen types are char, void (yes, it's actually used too), bool and all intN_t and uintN_t.

The last thing to do is to create protocol that could serialize such path type and value pack them into stream of bytes and send them (possibly over a network).

The current protocol is very simple. Each packet consists of destination (possibly broadcast) and type then there is path identifying object in an application or a backend and a data payload. The data payload consist in most of the cases of serialized C data types. Note that the protocol itself is in a proof of concept state as it lacks checksums and sizes in packet descriptions. More stable and less error prone protocol may be implemented later.

To sum it up an application exports a tree like structure of named typed objects some of them could be get and some of them set via messages send over a socket. Setting or getting such value may trigger an action from within the callback.

3.3.2 Backend part

The problems to be solved in backend are a little different than the application ones. Still there is a need for serialization both on the way from application as we need the widget properties to create UI and on the way from backend to application to change widget properties. The situation though is different as while application has all the widget attributes stored in its memory in the corresponding widgets the backend needs to load the application data structure and to store it somewhere.

To solve this problem a cache was created. For each application the backend maintains a cache which is tree-like structure that mirrors the structure of exported objects in an application. The values in the cache are again stored in native C programming language format and once resolved could be used directly from within the C code. The cache also allows us to store backend related data into its structure (for example you may want to store cursor position in textbox). Identifiers for backend objects inserted into this structure, by a convention, starts with dot which is a parallel to hidden files.

3.3.3 Backend and Application interaction

When an application starts, it connects to the application multiplexer. The multiplexer sends notification about new application to all backends.

When a backend starts, it connects to the application multiplexer and gets list of currently running applications.

When a backend decides to show an application it sends a request to the application (keep in mind that this request as well as reply travels through the multiplexer) and the application sends its exported tree to the backend. The backend could, by a similar request, ask for the layout rendering hints. Once backend gathers this information needed to show the application GUI it computes layout for all widgets in the application window. The resulting layout is computed from application structure, layout hints, currently used font etc. There may be one additional step once widget sizes and placements are computed. The canvas widget size may depend on the available size in the application window so its content may be, in some cases, retrieved after this step. Finally when all the data are ready the application GUI window is drawn on the screen and presented to the user.

Once application state changes, either because some user has clicked on a widget or just because application updates information about the task state, packet containing the changes is broadcasted (from within the application). Backends that are not showing this application interface (the application tree cache for such application doesn't exist) ignore such broadcasts. Backends that shows the application gets the notification and corresponding part of UI is updated possibly graphical representation of widget is redrawn.

3.3.4 GUI Layout description

As said before the Micro Witchcraft widget toolkit also includes means for describing the layout of widgets in a window. The description of a layout consists of data description and of an algorithm to compute actual sizes and positions.

The concepts and implementation are similar to some of the existing solutions, notable source of inspiration was TEX[10] typesetting system.

As mentioned before I've decided for dynamic font size driven layout rather than fixed one. Therefore the position of a widget is described in terms of grids and cells rather than coordinates and sizes.

The basic layout structure is a grid. Each grid composes of $N * M$ cells as it has N columns and M rows. Each cell could be occupied by a widget or by a grid (often called a subgrid regarding to the grid it does belong to). The grid also composes of margins, that are used to describe white space between columns and rows. There is $N + 1$ vertical margins and $M + 1$ horizontal margins (two outer margins of each kind and $N - 1$ respectively $M - 1$ inner margins). The structure, in terms of implementation is linearized in one dimensional array of structures starting with $N + M + 2$ margins followed by $N * M$ widgets and/or subgrids.

Each element of the grid has a stretchability defined, that is a number which controls allocation of the space for the particular element. Widgets and subgrids do have both horizontal and vertical stretch. The margins simply have only stretch as their direction is explicitly defined by the margin type (which is either horizontal or vertical).

The Layout Algorithm is two pass and recursive. Vertical and horizontal calculations are independent although computed at the same time. Moreover the vertical and horizontal calculations are exactly same, the only difference is that in the results describes different direction.

The first pass is used to compute minimum grid size. The algorithm starts from the root grid, that describes layout of the window. First minimal size needed for the margins is calculated, the size of the margin could be zero or margin size as defined by rendering hints. Once the space needed for horizontal and vertical margins is computed, we start to compute minimal size for the elements of the grid. Each widget has a function that returns minimal widget size in pixels and the result depends on currently chosen font, widget type, attributes etc. By calling this function we get minimal width and height for each widget in grid. The subgrids are computed recursively (minimal size of subgrid is minimal size of subgrid margins plus size needed for subgrid elements). Currently the grid can work in one of two modes. If grid is set to use FIT algorithm the size for certain column (row) is computed as a maximum from all minimal widths (heights) of elements in that particular column (row). If grid is set to HOMOGENEOUS each column (row) is set to maximum width (height) over all grid elements. This pass also computes overall horizontal and vertical stretchability for each grid, which is defined as a sum of maximums of stretch coefficients over the columns and rows. Note that we do not care about positions in this pass at all. The only information we get from this run a set of minimal sizes and sums of stretch factors.

The second pass of the algorithm is used to distribute the remaining free space and to compute widget placements. Before the second pass begins it must

[10] `http://en.wikipedia.org/wiki/TeX`

be given width and height of available space to fit the root grid to. The free space is then calculated as a difference of given space and minimum grid size computed by the first pass of the algorithm. If given space is smaller than minimal size, the positions are calculated for minimal size instead (possibly even for one of the directions). The algorithm starts with coordinates of the left top corner of the grid. Then in each step the coordinates are advanced as we go through the grid to the right and down rendering positions and sizes for each grid element.

Let's look closer at the steps of the algorithm. We start with the coordinates of the top left corner of the grid. Now we need to advance by the size of left and top margin, that size is counted as static size of the margin (either zero or margin size as defined by rendering hints) plus dynamic part defined by the stretch factor and available space. The dynamic part is counted as follows: if there is available space, it's multiplexed by stretch factor for the particular element and divided by a sum of stretch factors over corresponding direction. Now we have coordinates for a first widget/subgrid in the table. We count it's size that is, in both directions, similarly defined by a static part which is minimal widget/subgrid size and by the dynamic part defined by the horizontal and vertical stretch factors. The differences between the subgrid and widget, in this step of algorithm, are as follows: Firstly the source of minimal size is different and secondly the algorithm is called recursively to calculate positions and sizes for the subgrid before advancing to the next widget. Once we advance to the end of the row we simply reset the x coordinate to point to the end of the left outer vertical margin and advance vertical direction by a row height and corresponding inner vertical margin size. This steps are repeated until we advance to the last element in root grid and at that time coordinates and sizes for all widgets are computed.

The last thing to describe is how are widgets and subgrids placed into the grid cell. As defined in the previous paragraph the available space for a widget grid is at least as big as minimum grid size so each grid cell is at least as big as minimal widget/subgrid size. The widget/subgrid could be placed into the cell ether by positioning it to left/center/right in horizontal direction and top/center/bottom in vertical direction, or fit in one or both directions which causes widget to allocate all available space.

The result of the algorithm is an array of structures, each structure holds pointer to a widget representation (pointer somewhere into the tree like backend cache in our case) and a position and size.

The computational complexity of the algorithm is linear to the overall number of widgets and grids. In each pass we touch each grid element (be it margin, widget or subgrid) exactly once. Doing two passes the result should be multiplied by two. Lets W denotes number of widgets and G number of grids. Now upper bound on the number of margins is:

$$4 * (W + G - 1)$$

As we count four margins that directly touch each grid or widget, we may have counted some of them several times, which makes this number upper bound. Now we can state that the complexity is, in worst case:

$$O(W + G) = 2 * (4 * (W + G - 1) + W + G)$$

Which could be simplified to:

$$O(N) = 10 * N - 8$$

Where N is sum of widgets and grids.

3.4 Neko backend renderer

The Neko GUI render is intended to run as a user interface on a small[11] devices. The design goals are fitted for the intended purpose. It has fairly small memory footprint and all windows are rendered fullscreen or rather using all available space regarding the render image buffer, which fairly simplifies the window management code. Neko also has a panel for indicators such as battery charge, CPU usage, screen rotation, etc. as well as on-screen keyboard and virtual screens.

The internal design of the render is straightforward. There are two sources of events that change the render state. First one are applications. Applications sends events about change in widget values or in a window layout (currently window could be opened or closed). The second source of events are users, they press buttons on a keyboard and move and click with the mouse in order to change the widgets values. The Neko main loop revolves around these two sources of events.

When Neko is started, it initializes the graphical backend (which may be a Framebuffer on a PDA running Linux or a window on a desktop). Then it connects to the application multiplexer and gets a list of running applications. In order to render an application GUI the user must choose application (which is done by arrows and enter keys). Once application is selected, Neko requests the application to send it's widgets structure (Called CSI data in the source code) and window layout hints (only one window could be shown at the time for now). The structure and layout hints are cached and layout algorithm is used to compute widget positions and sizes. Then finally application GUI is presented to the user. Once application is visible on the screen, Neko uses incremental updates for the CSI data and/or layout which are send by an application to update the state of GUI (which may yield into recalling the layout algorithm). Currently such update redraws most of the widgets in the layout which could be and will be improved later.

As already said, the GUI rendering sizes are driven by a size of a font. The Neko render supports True Type fonts with arbitrary sizes so the rendering part of the backend is fully dynamic. All the GUI presented to the user, the panel with indicators, the on-screen keyboard and the actual application adjusts it's sizes automatically to the given font and its size. Although there is not yet a knob to adjust the font settings at runtime (currently the configuration is loaded from a file) the render code structure is written with this functionality in mind.

3.5 GFXprim library

The rendering part of Neko needed a graphical library or a set of libraries to handle bitmap manipulations as well as the drawing. In early stages this was

[11] Such as PDAs, smartphones, embedded devices or tables.

done by the SDL library[12], as this is defacto standard for writing graphical applications in Linux. Note that there are several graphics libraries available, for an example imlib2[13] however these are closely tied to the X server data structures and basically unusable for standalone Framebuffer application. Unfortunately the chosen SDL library, or rather the family of libraries with SDL prefix, had several shortcomings.

The SDL is trying to abstract the operating system and is shadowing details needed to write correct application. This was the most problematic one. The SDL is hiding file descriptors or underlying format of display. The computer, as seen by SDL library, is one monitor with one keyboard and mouse, it has no reasonable support for touchscreen and other newer input devices.

Then there are less problematic bugs in the surrounding libraries as in the SDL_gfx which couldn't draw even basic geometric shapes right. So rather than trying to work around the problems in SDL the GFXprim library was started. First it just filled the gaps or recreated similar functionality in more sane manner, still using parts of the SDL. And finally after more than two years of development GFXprim yielded into a library that provides more than complete API for writing graphics applications.

3.5.1 GFXprim library description

The project frontpage says that *GFXprim is Open-source modular 2D bitmap graphics library with emphasis on speed and correctness*, which more or less summarizes the project goals. One of the key points of the library is templating. Most of the bitmap manipulation code is written in Python templating language called jinja2[14] which is used to generate code accordingly to the pixel format descriptions from the configuration file. The library itself is modular and so the code is divided into several parts accordingly to the functionality it implements.

Core implements basic data structures like the GP_Context that describes in-memory bitmaps. It includes generated inline functions for GetPixel and PutPixel operations, bitmap blitting, gamma correction code, debug message infrastructure and more.

Backends implement means for drawing on the screen or a window and getting input events such as keys presses or pointer positions. There is a separate input layer, that backends use to deliver events to the application. Backends are split into drawing and input part because in real world input devices do not necessarily corresponds to the display you are drawing to. So you are free, for an example, to use Linux input character device (such as `/dev/input/event0`) when drawing on Framebuffer instead of the legacy kbd driver). Current backends do support Linux Framebuffer, X server and SDL as a graphics backend.

Gfx part of the library implements, as the name suggests, graphics operations such as drawing lines, circles, polygons etc. This is the oldest part of the library

[12] `http://www.libsdl.org/`

[13] `http://docs.enlightenment.org/api/imlib2/html/`

[14] `http://jinja.pocoo.org/`

which gave it its name.

Text part of the library is used to render text. It contains two compiled-in fonts as well as support for True Type fonts using FreeType[15] library.

Loaders provides API to load and save images files of several formats. Loaders utilize standard libraries for complex formats such as jpeg, png and gif while support for less complicated formats (such as bmp and pbm) is written from scratch.

Filters are part of the library I'm personally proud of. Filters implement fairly fast and correct algorithms for various task such as resampling, convolutions, dithering, image restoration and more. Most of the code is implemented using integer arithmetics which is specially fast on small devices where floating point is commonly emulated. Majority of the filters are written using jinja2 templating engine, so creating set of filters for new pixel format is just matter of adding pixel description into the configuration file and typing: '`make clean && make`'. Support for multi-threaded filters is in the works at the time of writing this document.

Lastly but not least I would like to thank two of my friends for helping with the library development. Namely to Jiri 'Bluebear' Dluhos who wrote most of the code in the gfx part and to Tomas Gavenciak who helped with the templating engine, the code generation and with the Python bindings.

For more information about the library, its API, to ask question on the mailing list, or to checkout latest source code follow one of the links on the library webpage at `http://gfxprim.ucw.cz`.

[15] `http://www.freetype.org/index2.html`

4. Conclusion

The goal of this thesis was to discuss and implement a widget toolkit, which in my opinion was successful. On the other hand the result is not yet usable for common users, although it could be successfully started on a PDA (and actually runs on my old Sharp Zaurus) and despite the fact that there are several applications ready (app for playing music and app for reading pdf documents to name the most useful ones). There are different reasons for this.

The most problematic part is the network serialization protocol. As stated before the current protocol is in proof-of-concept state. It's not effective for large chunks of data and is missing means for recovering from errors. Solution for this problem is to redesign the network protocol.

Then there are easier problems to solve. For example there are no means to start an application besides starting it manually from a console. More generally there are still some rough edges that needs to be taken care of before the result is usable.

List of Abbreviations

API	Application Interface
CLI	Command Line Interface
CPU	Central Processing Unit
DPI	Dots per Inch
GUI	Graphics User Interface
PDA	Personal Digital Assistant
SDL	Simple DirectMedia Layer library
TEX	TEX Typesetting System
TUI	Text User Interface
UI	User Interface
UNIX	The UNIX Operating System

A. Compilation and Installation

The Micro Witchcraft library can be compiled and installed on any reasonable modern Linux distribution (Most of the testing was done on Gentoo[1] and Debian[2] on x86, x86_64, arm and ppc). Majority of the code is portable to any POSIX[3] compatible UNIX however no testing was done to ensure the portability. To compile Micro Witchcraft GFXprim library must be compiled and installed first.

A.1 GFXprim Compilation and Installation

The GFXprim library needs following utils to be compiled:

- **C compiler** The build was tested with gcc 4.1.2[4] and newer or with clang 3.0[5] and newer.
- **GNU make** The GNU make 3.81[6] or newer.
- **libc devel** The devel package for C library.
- **Python** Python programming language 2.5 or newer (including 3.0)[7].
- **jinja2** Jinja2 templating engine[8].

Optional but still important devel libraries are:

- **libpng** Libpng[9] library to support PNG images.
- **libjpeg** Libjpeg[10] library to support JPEG images.
- **giflib** Giflib[11] library to support GIF images.
- **libX11** Xlib[12] interface to X Window System Protocol.
- **FreeType** FreeType[13] library to support TrueType font rendering.

The build process is standard and pretty straightforward, basically the same as for any UNIX software. After unpacking the tarball (or checking latest source from the git repository) the `configure` script must be executed first.

Note that git checkouts in contrast with the attached tarball needs either **check**[14] unit text framework to be installed or the test target from the main Makefile must be disabled.

[1] `http://www.gentoo.org/`
[2] `http://www.debian.org/`
[3] `http://en.wikipedia.org/wiki/POSIX`
[4] `http://gcc.gnu.org/`
[5] `http://clang.llvm.org/`
[6] `http://www.gnu.org/software/make/`
[7] `http://www.python.org/`
[8] `http://jinja.pocoo.org/docs/`
[9] `http://www.libpng.org/pub/png/`
[10] `http://www.ijg.org/`
[11] `http://giflib.sourceforge.net/`
[12] `http://www.freedesktop.org/wiki/Software/X11`
[13] `http://www.freetype.org/`
[14] `http://sourceforge.net/projects/check/`

Example configure script output:

```
sh$ ./configure
Basic checks
------------
Checking for working compiler (gcc) ... Yes
Checking for python module jinja2 ... Yes
Checking for working swig ... Yes

Checking for libraries
----------------------
Checking for '/usr/include/png.h' ... Yes
Checking for '/usr/include/SDL/SDL.h' ... Yes
Checking for '/usr/include/jpeglib.h' ... Yes
Checking for '/usr/include/gif_lib.h' ... Yes
Checking for '/usr/include/X11/Xlib.h' ... Yes
Checking for '/usr/include/ft2build.h' ... Yes
Checking for '/usr/include/linux/videodev2.h' ... Yes

Libraries to link against
-------------------------
    libpng : Enabled
             - Portable Network Graphics Library

    libsdl : Enabled
             - Simple Direct Media Layer

      jpeg : Enabled
             - Library to load, handle and manipulate images in the JPEG format

    giflib : Enabled
             - Library to handle, display and manipulate GIF images

    libX11 : Enabled
             - X11 library

  freetype : Enabled
             - A high-quality and portable font engine

      V4L2 : Enabled
             - Video for linux 2

Config 'config.h' written
Config 'config.gen.mk' written
```

Then typing `make` or `make -jN` where N is number of jobs running simultaneously will build the library and typing `make install` with sufficient privileges will install the headers into `/usr/include/GP/` and libraries into `/usr/lib/` or `/usr/lib64/`.

A.2 Micro Witchcraft Compilation

Once the GFXprim library is installed the Micro Witchcraft could be finally build. The build process of the library is yet a little spartan. There is no configure script and devel libraries must be installed before the compilation.
The utils to build Micro Witchcraft are:

- **C compiler** The build was tested with gcc 4.1.2[15] and newer or with clang 3.0[16] and newer.
- **GNU make** The GNU make 3.81[17] or newer.
- **libc devel** The devel package for C library.

The devel libraries among the GFXprim library are:

- **libasound** ALSA library[18] (needed for sound mixer and MP3 player).
- **libmpg123** MP3 decoding library[19] (needed for MP3 player).

There is a pdf viewer, which is not build by default, that needs libfitz library which needs to be checked out from git, patched, compiled and installed (See README in the viewer directory).

Then again typing `make` should build the library and put resulting libraries and binaries into `build/` directory. The binaries are, to ease the development phase, statically linked with the Micro Witchcraft libraries.

Once binaries are compiled, the Appd, Neko render and Application could be started just by running corresponding binaries. The Neko render needs a config file to be created see `apps/backends/neko/data/README` for details. Also note that both applications and backends do connect to the Appd multiplexer and as a such it must be started first.

[15] `http://gcc.gnu.org/`
[16] `http://clang.llvm.org/`
[17] `http://www.gnu.org/software/make/`
[18] `http://www.alsa-project.org/alsa-doc/alsa-lib/`
[19] `http://mpg123.org/api/`

B. API Description

Following pages describes some of the Micro Witchcraft API. For more comprehensive description go to the the project pages at `http://metan.ucw.cz/MW/`.

B.1 Minimal Application Example

Following code implement fairly minimal application which would show a window with centered "Hello World!" text. The widget layout is defined and initialized statically. The `main()` routine connects to the multiplexer, creates a window with one label widget and a layout and waits for any events.

```
#include <MW.h>

static struct MW_WidgetGrid win_grid = {
        .cols = 1, .rows = 1,
        .elems = {
         /* horizontal margins from left to right */
         {.type = MW_GRID_MARGIN, .stretch = 1},
         {.type = MW_GRID_MARGIN, .stretch = 1},
         /* vertical margins from up to down */
         {.type = MW_GRID_MARGIN, .stretch = 1},
         {.type = MW_GRID_MARGIN, .stretch = 1},
         /* widgets/subboxes */
         {.type = MW_GRID_WIDGET, .id = "label1", .variant = MW_WV_FIT},
        }
};

int main(int argc, char *argv[])
{
        struct MW_Window *win;
        struct MW_Widget *wid;

        if (!MW_AppInit("Hello World", &argc, &argv)) {
                MW_MSG_INFO("Failed to initalize application");
                MW_AppExit(1);
        }

        wid = MW_WidgetLabelCreate("label1", 0, "Hello World!");
        win = MW_WindowCreate("win", "Hello World App", wid);

        MW_WindowGridSet(win, &win_grid);
        MW_WindowShow(win);

        MW_AppMainLoop();
}
```

The CSI widget structure of the window as shown by the **MW_AppdShell** backend:

```
NOTICE: App 'Hello World:13' connected
> ls
Applications (1)
----------------------------------------
app: 'Hello World:13'
----------------------------------------
> cd 13
NOTICE: App 'Hello World:13' is not loaded, sending CSI request
NOTICE: CSI for App 'Hello World:13' loaded successfully
'Hello World:13'> ls
CSICache {
 group 'win' {
  group 'label1' {
   max_size:uint32 = 12
   label:string = 'Hello World!'
   type:uint8 = 4
  }
  label:string = 'Hello World App'
  show:bool = true
 }
}
'Hello World:13'>
```

B.2 Application

MW_AppInit – *Widget API*

---

bool `MW_AppInit(`**const char** `*name,` **int** `*argc,` **char** `**argv[]);`

DESCRIPTION
Connects application to the Appd multiplexer and initializes Application internal data structures.

Function must be called before any other widget API is used.

The argc and argv parameters may be used for changing connection destination and port. The format is -c machine[:port]. If the -c parameter is encountered first it's parsed and argc and argv are shifted so they point to the first parameter after it.

RETURN VALUE
Function returns true if connection to Appd was successful otherwise false is returned.

MW_AppExit – *Widget API*

---

void `MW_AppExit(int exit_code);`

DESCRIPTION
Disconnect application from Appd, do all cleanups and calls exit(exit_code).

RETURN VALUE
Function does not return i.e. stops the program execution.

MW_AppMainLoop – *Widget API*

---

void `MW_AppMainLoop(void);`

DESCRIPTION
Start an application loop. From this point on, the application waits for events.

These events include:

- Events from backend(s)
- Application timer
- Any file descriptors added via MW_AppWatchFd(3)

RETURN VALUE
This function does not return.

MW_AppPoll – *Widget API*

---

void `MW_AppPoll(void);`

DESCRIPTION
Does the same as MW_AppMainLoop(3) but returns:

- if there are no events to be processed
- right after current events are processed

So this function should be called periodically.

Note that the app timer precision depends on how often is this function called.

Generally you should use MW_AppMainLoop(3) instead whenever possible.

MW_AppTimerStart – *Widget API*

```
void MW_AppTimerStart(uint32_t (*Callback)(void *priv), void *priv,
                      uint32_t msec);
```

DESCRIPTION
Starts an application timer. The timer is implemented using the SIGALRM signal so mixing it with alarm(3) or sleep(3) may cause unexpected results.

Once the timer has expired, flag is set and MW_AppMainLoop(3) is interrupted and timer callback is called. The callback is not called from whiting the signal handler so it's completely safe to call signal-async-unsafe functions.

The return value from the timer callback determines interval for the next timer and returning zero stops the timer.

MW_AppTimerStop – *Widget API*

```
void MW_AppTimerStop(void);
```

DESCRIPTION
Stops the timer execution.

The signal for next timer is not aborted may arrive but is ignored.

MW_AppWatchFd – *Widget API*

```
void MW_AppWatchFd(struct MW_Fd *fd);
```

DESCRIPTION
Adds Fd to the application Fd queue. See MW_FdQueue(3) for details.

MW_AppWindowAttach – *Widget API*

```
void MW_AppWindowAttach(struct MW_Window *self);
```

DESCRIPTION
Adds window into the list of application windows.

MW_AppWindowById – *Widget API*

```
struct MW_Window *MW_AppWindowById(const char *id);
```

DESCRIPTION
Looks up application window by its id.

RETURN VALUE
Returns pointer to resolved window or NULL if there is no window with such id.

MW_AppWindowRemove – *Widget API*

```
void MW_AppWindowRemove(struct MW_Window *self);
```

DESCRIPTION
Removes a window from the list of application windows.

MW_AppWindowRemoveById – *Widget API*

```
struct MW_Window *MW_AppWindowRemoveById(const char *id);
```

DESCRIPTION
The same as MW_AppWindowRemove(3) but the application window is looked up by its id and pointer to the window is returned.

RETURN VALUE
Returns pointer to a removed window or NULL if there were no window with such id.

B.3 Window

MW_WindowAlloc – *Widget API*

```
struct MW_Window *MW_WindowAlloc(const char *id, const char *label, ...);
```

DESCRIPTION
Does exactly the same as MW_WindowCreate(3) but the resulting window is not automatically attached to the application.

RETURN VALUE
Function returns pointer to allocated and initialized MW_Window structure or NULL in case of malloc(2) failure.

MW_WindowByIdWidgetAdd – *Widget API*

```
void MW_WindowByIdWidgetAdd(const char *id, struct MW_Widget *wid);
```

DESCRIPTION
Does the same as MW_WindowWidgetAdd(3) except the pointer to the window is resolved from window id. This doesn't work for windows that are removed from application.

MW_WindowByIdWidgetsAdd – *Widget API*

```
void MW_WindowByIdWidgetsAdd(const char *id, ...);
```

DESCRIPTION
Combination of MW_WindowWidgetsAdd(3) and MW_WindowByIdWidgetAdd(3).

MW_WindowCreate – *Widget API*

```
struct MW_Window *MW_WindowCreate(const char *id, const char *label, ...);
```

DESCRIPTION
Allocates and initializes struct MW_Window and attaches the window to application.

The id string must be unique across all application windows and also must not contain colon nor start with dot. Later the window pointer can be resolved from the id string.

The window label is a string that may be shown in the user interface.

You can additionally pass some number (including none) of widgets that would be added into the window. See MW_WindowWidgetAdd(3) for more details.

The window is automatically attached to application windows. If you just need to prepare a window for attaching it later use MW_WindowAlloc(3).

RETURN VALUE
Function returns pointer to allocated and initialized MW_Window structure or NULL in case of malloc(2) failure.

MW_WindowDestroy – *Widget API*

```
void MW_WindowDestroy(struct MW_Window *self);
```

DESCRIPTION
Removes window from app window list, calls destructors of all widgets associated with window, sends notification to Appd that window was closed, and frees memory used by the window.

RETURN VALUE
None.

MW_WindowGridGet – *Widget API*

```
struct MW_WidgetGrid *MW_WindowGridGet(struct MW_Window *self);
```

DESCRIPTION
Returns a pointer to the widget grid associated with a window.

RETURN VALUE
If window has no grid associated yet, NULL is returned otherwise pointer to the window widget grid is returned.

MW_WindowGridSet – *Widget API*

```
void MW_WindowGridSet(struct MW_Window *self, struct MW_WidgetGrid *grid);
```

DESCRIPTION
Sets a widget grid for window. The pointer to grid is stored in the window structure and in case application is connected to appd a notification that grid has changed is broadcasted to backends.

MW_WindowLabelGet – *Widget API*

```
const char *MW_WindowLabelGet(struct MW_Window *self);
```

RETURN VALUE
Returns window label.

MW_WindowLabelSet – *Widget API*

```
bool MW_WindowLabelSet(struct MW_Window *self, const char *label);
```

DESCRIPTION
Sets window label. If the label is longer than MW_WINDOW_LABEL_MAX it's shortened to fit.

RETURN VALUE
Returns true if label was shorter than MW_WINDOW_LABEL_MAX, false otherwise.

MW_WindowWidgetAdd – *Widget API*

```
void MW_WindowWidgetAdd(struct MW_Window *self, struct MW_Widget *widget);
```

DESCRIPTION
Adds widget into window list of widgets.

MW_WindowWidgetById – *Widget API*

```
struct MW_Widget *MW_WindowWidgetById(struct MW_Window *self, const char *id);
```

DESCRIPTION
Look for a widget by an id in a window.

RETURN VALUE
Function returns pointer to resolved widget or NULL if not found.

MW_WindowWidgetsAdd – *Widget API*

```
void MW_WindowWidgetsAdd(struct MW_Window *self, ...);
```

DESCRIPTION
Does the same as MW_WindowWidgetAdd(3) only takes some number of widgets as parameters (including none).

B.4 Widget Grid

MW_WidgetGrid – *Widget API*

```
enum MW_WidgetGridType {
      /*
       * The minimum size of each column/row is fit to the biggest widget in
       * the corresponding column/row. Various columns/rows can have different
       * minimum size.
       */
         MW_WIDGET_GRID_FIT = 0x00,

      /*
       * The widget grid is homogeneous, each cell (element) has the same
       * minimum size which is equal to the minimum size of the biggest
       * element in grid cell. All columns/rows have the same minimum size.
       */
         MW_WIDGET_GRID_HOMOGENEOUS = 0x01,
};

struct MW_WidgetGrid {
         /* numbers of cols/rows in the grid */
         uint8_t cols;
         uint8_t rows;

         /* type == enum MW_WidgetGridType */
         uint8_t type;

         ...

      /*
       * Sequence of:
       *
       * col + 1   horizontal margins
       * row + 1   vertical margins
       * col * row widgets/subgrids
       */
         struct MW_WidgetGridElem elems[];
};
```

DESCRIPTION

A widget grid is a grid-like structure that describes widget layout in a window.

The array of MW_WidgetGridElem(3) elements consists of (col + 1) horizontal margins, (row + 1) vertical margins and finally of (col * row) elements, either widgets or subgrids.

The horizontal and vertical stretching coefficients describe the stretchability of the grid as a widget. They are only used if the grid is a part of another container.

The type describes space allocation behavior. The MW_WIDGET_GRID_FIT allocates grid space to so that particular row, col is fit to the biggest grid element in it. On the contrary, MW_WIDGET_GRID_HOMOGENEOUS finds the maximum for the whole grid and sets all grid cells to it's size.

The widget grid can either be created and initialized statically (as shown in the example) or created dynamically by MW_WidgetGridCreate(3) and there are a few helper functions for setting it's parameters.

The grid size is calculated in two steps. In the first step, the minimum size of the grid is determined. For each grid element, minimum size is computed and depending on grid type, the minimum sizes for columns and rows are computed. This is done recursively for subgrids, first computing the minimum size for the lowest level of grids and computing higher ones while returning from the recursion. Also, margin sizes are taken into an account in this step. After this step, each grid has its minimum size for its cells and margins known, along with maximal stretch for each column and margin (each grid element has defined horizontal and vertical stretch factor).

In the second step, the available space is distributed accordingly to the stretch factors. The rendering backend passes a window size so available space is computed as window size minus minimal size. If available space is greater than zero the space is distributed between grid margins and cells accordingly to stretch factors. The stretch factor equal to zero means don't allocate space at all.

Note that the calculations for horizontal and vertical direction are done independently. So an application may be squeezed in one direction to its minimum size while the other direction could have twice of the minimum, depending on window size and grid properties.

EXAMPLE

```
struct MW_WidgetGrid sub_grid[] = {
        ...
};

struct MW_WidgetGrid win_grid[] = {
        .rows = 1, cols = 3,
```

```
        .elems = {
         /* cols + 1 horizontal margins from left to right */
         {.type = MW_GRID_MARGIN},
         {.type = MW_GRID_MARGIN},
         {.type = MW_GRID_MARGIN},
         {.type = MW_GRID_MARGIN},

         /* rows + 1 vertical margins from up to down */
         {.type = MW_GRID_MARGIN, .stretch = 1},
         {.type = MW_GRID_MARGIN, .stretch = 1},

         /* rows * cols elems */
         {.type = MW_GRID_WIDGET, .id = "foo"},
         {.type = MW_GRID_WIDGET, .id = "bar"},
         {.type = MW_GRID_SUBGRID, .grid = &subgrid},
        }
};
```

MW_WidgetGridDestroy – *Widget API*

```
void MW_WidgetGridDestroy(struct MW_WidgetGrid *self);
```

DESCRIPTION

Destroys (frees memory) for widget grid including all subgrids recursively.

```
/*
 * Widget grid element type.
 */
enum MW_WidgetGridElemType {
        /* widget */
        MW_GRID_WIDGET,
        /* subgrid */
        MW_GRID_SUBGRID,
        /* margins */
        MW_GRID_NONE,
        MW_GRID_MARGIN,
};

/*
 * Alignment flags.
 */
enum MW_WG_Position {
        /* Horizontal */
        MW_WG_HCENTER = 0x00,
        MW_WG_LEFT    = 0x01,
        MW_WG_RIGHT   = 0x02,
        MW_WG_HFILL   = 0x04,
        /* Vertical */
        MW_WG_VCENTER = 0x00,
        MW_WG_TOP     = 0x10,
        MW_WG_BOTTOM  = 0x20,
        MW_WG_VFILL   = 0x40,
        /* Combined/Shortcuts */
        MW_WG_CENTER  = MW_WG_HCENTER | MW_WG_VCENTER,
        MW_WG_FILL    = MW_WG_HFILL | MW_WG_VFILL,
};

#define MW_WG_H(x) ((x) & 0x07)
#define MW_WG_V(x) ((x) & 0x70)

struct MW_WidgetGridElem {
        /* element type widget/subgrid/margin */
        uint8_t type;

        /* margin stretch */
        uint8_t stretch;

        /* widget/subgrid alignment flags */
        uint8_t align;

        /* widget variant */
        MW_WidgetVariant variant;
```

```
        /* widget id */
        char id[MW_ID_MAX];

        /* subgrid pointer */
        struct MW_WidgetGrid *subgrid;
};
```

DESCRIPTION
The widget grid element is part of MW_WidgetGrid(3) structure. It's used to describe either margins between elements in grid, or grid elements (widgets or subgrids).

If 'element' is a margin, either horizontal or vertical, the behavior is as follows. The element type describes static part of the margin, where MW_GRID_NONE adds no size and MW_GRID_MARGIN adds fixed margin size (as defined by the rendering backend). The stretch factor defines how aggressively the margin 'eats' available space. The stretch factor set to zero means don't 'eat' any space at all, for all non-zero values the stretch factor defines how much of available space would be allocated for particular grid cell or margin.

When 'element' is a widget, ie. type is set to MW_GRID_WIDGET, the layout is defined by 'id', 'variant' and 'align'. 'id' is an ID of the widget (in this window) to be rendered into this grid element. 'variant' together with widget type defines how the widget is rendered on the screen. For example, the integer widget type has possible variants MW_WV_HSLIDER, MW_WV_VSLIDER, MW_WV_KNOB and MW_WV_SPIN_BUTTON. The last parameter is 'align' that defines the alignment of the widget in available space.

When 'element' is a subgrid, the 'subgrid' and 'align' fields should be set.

MW_WidgetGridElemGet – *Widget API*

```
struct MW_WidgetGridElem *MW_WidgetGridElemGet(struct MW_WidgetGrid *self,
                                               uint8_t col, uint8_t row);
```

DESCRIPTION
Returns pointer to MW_WidgetGridElem(3) occupying grid cell on column col and row row (i.e. Widget or Subgrid).

MW_WidgetGridHMarginSet – *Widget API*

```
void MW_WidgetGridHMarginSet(struct MW_WidgetGrid *self,
                             uint8_t row, uint8_t type, uint8_t stretch);
```

DESCRIPTION
Sets the properties of the horizontal margin of the given row of the grid.

There is (grid->rows + 1) margins in total (two outer margins and (grid->rows-1) inner margins). Margins are indexed from 0 to grid->rows.

The type could be either MW_GRID_NONE or MW_GRID_MARGIN.

MW_WidgetGridHMarginsSet – *Widget API*

```
void MW_WidgetGridHMarginsSet(struct MW_WidgetGrid *self,
                              uint8_t type, uint8_t stretch);
```

DESCRIPTION
Sets all grid horizontal margins.

The type could be either MW_GRID_NONE or MW_GRID_MARGIN.

MW_WidgetGridPrint – *Widget API*

```
void MW_WidgetGridPrint(struct MW_WidgetGrid *self);
```

DESCRIPTION
Prints human-readable dump of widget grid into stdout (including subgrids). Useful for debugging.

MW_WidgetGridVMarginSet – *Widget API*

```
void MW_WidgetGridVMarginSet(struct MW_WidgetGrid *self,
                             uint8_t col, uint8_t type, uint8_t stretch);
```

DESCRIPTION
Sets properties of a grid vertical margin on the given column. There is (grid->cols + 1) vertical margins in total (two outer margins and (grid->cols-1) inner margins). Margins are indexed from zero to grid->cols.

The type could be either MW_GRID_NONE or MW_GRID_MARGIN.

MW_WidgetGridVMarginsSet – *Widget API*

```
void MW_WidgetGridVMarginsSet(struct MW_WidgetGrid *self,
                              uint8_t type, uint8_t stretch);
```

DESCRIPTION
Sets all grid vertical margins.

The type could be either MW_GRID_NONE or MW_GRID_MARGIN.

B.5 Button Widget

MW_WidgetButtonCallbackSet – *Widget API*

```
void MW_WidgetButtonCallbackSet(struct MW_Widget *self,
                                void (*Callback)(struct MW_Widget *self),
                                void *callback_priv);
```

DESCRIPTION
Sets a widget button Callback and callback private pointer. The Callback is called once button is pressed (either by an event from backend or by calling MW_WidgetButtonPress(3) or MW_WidgetButtonStateSet(3)).

The callback_priv pointer is for your use it's saved and is accessible inside of the Callback context by calling MW_WidgetCallbackPrivGet(3).

MW_WidgetButtonCreate – *Widget API*

```
struct MW_Widget *MW_WidgetButtonCreate(const char *id,
                                        enum MW_WidgetButtonType type,
                                        const char *label);
```

DESCRIPTION
Button is an abstraction for widget that, when activated, calls callback. Basically it's element that triggers some action.

The id is widget id and must be unique inside window, must not contain colon and must not start with dot.

The button type hints the render about the type of action which button triggers and may be one of following:

- MW_WIDGET_BUTTON_LABELED
- MW_WIDGET_BUTTON_OK
- MW_WIDGET_BUTTON_CANCEL
- MW_WIDGET_BUTTON_ABORT
- MW_WIDGET_BUTTON_APPLY
- MW_WIDGET_BUTTON_YES
- MW_WIDGET_BUTTON_NO
- MW_WIDGET_BUTTON_OPEN
- MW_WIDGET_BUTTON_CLOSE
- MW_WIDGET_BUTTON_PREV
- MW_WIDGET_BUTTON_NEXT
- MW_WIDGET_BUTTON_START
- MW_WIDGET_BUTTON_STOP
- MW_WIDGET_BUTTON_PAUSE
- MW_WIDGET_BUTTON_RESET
- MW_WIDGET_BUTTON_ZOOM
- MW_WIDGET_BUTTON_ZOOM_IN
- MW_WIDGET_BUTTON_ZOOM_OUT

The MW_WIDGET_BUTTON_LABELED is an special type for which the label parameter allowed not to be NULL and rather label string used for the button label. In all other cases the widget label is generated from the type.

RETURN VALUE
Returns pointer to allocated and initialized widget or NULL in case of malloc(3) failure.

MW_WidgetButtonDestroy – *Widget API*

```
void MW_WidgetButtonDestroy(struct MW_Widget *self);
```

DESCRIPTION
Destroys an integer widget, if self is NULL no operation is performed.

MW_WidgetButtonPress – *Widget API*

```
void MW_WidgetButtonPress(struct MW_Widget *self);
```

DESCRIPTION
Presses a button. Calling this would press a button and would cause (if Callback is set) the Callback to be called.

MW_WidgetButtonPressed – *Widget API*

```
bool MW_WidgetButtonPressed(struct MW_Widget *self);
```

DESCRIPTION
Returns button state once the button is pressed it stays so until it's released by MW_WidgetButtonRelease(3).

RETURN VALUE
Returns true if button is pressed false otherwise.

MW_WidgetButtonRelease – *Widget API*

```
void MW_WidgetButtonRelease(struct MW_Widget *self);
```

DESCRIPTION
Release a button. Usually called after work triggered by a button was finished.

MW_WidgetButtonStateSet – *Widget API*

```
void MW_WidgetButtonStateSet(struct MW_Widget *self, bool state);
```

DESCRIPTION
Sets button state.

Calling this function with state true will press the button which will cause the button (if Callback is set) to call the Callback.

Calling it with state false will release the button.

MW_WidgetButtonTypeGet – *Widget API*

```
enum MW_WidgetButtonType MW_WidgetButtonTypeGet(struct MW_Widget *self);
```

RETURN VALUE
Returns button type.

MW_WidgetButtonTypeName – *Widget API*

```
const char *MW_WidgetButtonTypeName(enum MW_WidgetButtonType type);
```

RETURN VALUE
Returns button type name, a string.

B.6 Canvas Widget

MW_WidgetCanvasCreate – *Widget API*

```
/*
 * Structure that you get in the canvas drawing callback.
 */
struct MW_WidgetCanvas {
        /* Pointer to canvas buffer */
        GP_Context *canvas;
        /* Background color to use */
        GP_Pixel bg_color;
        /* User priv pointer */
        void *priv;
};

struct MW_Widget *MW_WidgetCanvasCreate(const char *id, uint32_t w, uint32_t h,
                                        void *priv,
                                        void (*Draw)(struct MW_WidgetCanvas *));
```

DESCRIPTION
Creates a canvas widget, that is a widget that represents a picture.

The id is widget id and must be unique inside window, must not contain colon and must not start with dot.

The w and h denotes size of the pixmap in pixels. If w and/or h is set to zero the pixmap is stretchable in the particular direction; that means that the widget fills available space. See MW_WidgetGrid(3) for details on space allocation.

The canvas widget doesn't hold any bitmap data at all. Instead of that the Draw callback is called once application requests the pixmap data. Note that depending on backend type the passed context would have different pixel type and if the canvas is stretchable, different size.

The priv pointer is used to store user pointer and is included in the structure passed to canvas Draw callback.

RETURN VALUE
Function returns pointer to allocated and initialized context widget or NULL in case of malloc(3) failure.

MW_WidgetCanvasDestroy – *Widget API*

```
void MW_WidgetCanvasDestroy(struct MW_Widget *self);
```

DESCRIPTION
Destroys a integer widget, if self is NULL no operation is performed.

MW_WidgetCanvasEventCallbackSet – *Widget API*

```
void MW_WidgetCanvasEventCallbackSet(struct MW_Widget *self,
                                     void (*Event)(struct MW_Widget *self,
                                                   GP_Event *ev),
                                     void *priv);
```

DESCRIPTION
Sets canvas event callback. The canvas widget, in contrast with other widgets, could receive 'raw' events (keys strokes, pointer event). The pointer events are normalized so that left upper pixel in the canvas has coordinates at (0,0).

MW_WidgetCanvasRedraw – *Widget API*

```
void MW_WidgetCanvasRedraw(struct MW_Widget *self);
```

DESCRIPTION
Send a canvas redraw event to backends. The canvas pixmap is usually cached at the backends, so once the image in your canvas becomes outdated and needs to be updated, call this function. It will send a notification that canvas needs to be redrawn.

Once the backend receives the the notification, it may, depending if particular application window is shown on the display, request the canvas rendering. Once application receives rendering request the canvas Draw callback is called which is exactly the point, where the actual image is rendered. Then the rendered image is send to the backend that had sent the request.

MW_WidgetCanvasResize – *Widget API*

```
void MW_WidgetCanvasResize(struct MW_Widget *self, uint32_t w, uint32_t h);
```

DESCRIPTION
Resizes the canvas pixmap. After calling this call the redraw request is sent automatically (as MW_WidgetCanvasRedraw(3) was called afterwise).

If w and/or h is set to zero the canvas fills available space in that particular direction. See MW_WidgetGrid(3) for details for space allocation.

B.7 Choice Set Widget

MW_WidgetChoiceSetCreate – *Widget API*

```
struct MW_Widget *MW_WidgetChoiceSetCreate(const char *id, const char *list[],
                                           uint32_t initial_sel);
```

DESCRIPTION
Creates an Choice Set widget, that is a widget that allows user to choose precisely one object from a set described by a list of strings.

The id is widget id and must be unique inside window, must not contain colon and must not start with dot.

The list is NULL terminated array of C strings.

The initial_sel is index for initial selection (starting from zero).

The widget callback, if set, is called once the chosen object is changed from any of the back ends. And isn't called upon calling MW_WidgetChoiceSetSelSet(3).

RETURN VALUE
Function returns pointer to allocated and initialized integer widget or NULL in case of malloc(3) failure.

MW_WidgetChoiceSetDestroy – *Widget API*

```
void MW_WidgetChoiceSetDestroy(struct MW_Widget *self);
```

DESCRIPTION
Destroys an Choice Set widget, if self is NULL no operation is performed.

MW_WidgetChoiceSetListGet – *Widget API*

```
const char **MW_WidgetChoiceSetListGet(struct MW_Widget *self);
```

RETURN VALUE
Returns pointer to, NULL terminated, array of C strings that represents the set of values to choose from.

MW_WidgetChoiceSetSelGet – *Widget API*

```
uint32_t MW_WidgetChoiceSetSelGet(struct MW_Widget *self);
```

RETURN VALUE
Returns currently selected choice (i.e. index of choice in list) starting with zero.

MW_WidgetChoiceSetSelSet – *Widget API*

```
void MW_WidgetChoiceSetSelSet(struct MW_Widget *self, uint32_t sel);
```

DESCRIPTION
Changes the currently selected choice. Unlike situation, where choice is set by a backend widget callback is not called upon calling this function.

MW_WidgetChoiceSetSizeGet – *Widget API*

```
uint32_t MW_WidgetChoiceSetSizeGet(struct MW_Widget *self);
```

RETURN VALUE
Returns size of the set, i.e. number of elements to choose from.

B.8 Integer Widget

MW_WidgetIntegerCreate – *Widget API*

```
enum MW_WidgetIntegerFlags {
        MW_WIDGET_INTEGER_OVERFLOW = 0x01,
};

struct MW_Widget *MW_WidgetIntegerCreate(const char *id,
                                         int32_t min, int32_t max,
                                         int32_t val, uint8_t flags,
                                         const char *label);
```

DESCRIPTION
Creates an integer widget, that is a widget with integer value between the min and max (including them).

The id is widget id and must be unique inside window, must not contain colon and must not start with dot.

Initially it's value is set to val.

The flags could be either 0 or MW_INTEGER_OVERFLOW which causes it automatically overflow to min when reaching value greater than max and vice versa.

The label is string that is shown in backend somewhere to describe widget's function. Could be NULL for no label.

RETURN VALUE
Function returns pointer to allocated and initialized integer widget or NULL in case of malloc(3) failure.

MW_WidgetIntegerDec – *Widget API*

```
void MW_WidgetIntegerDec(struct MW_Widget *self);
```

DESCRIPTION
Decrease widget integer value by one.

If resulting value would be lesser than minimum and MW_WIDGET_INTEGER_OVERFLOW was passed to MW_WidgetIntegerCreate(3), the value is set to maximum, otherwise it's no-op.

MW_WidgetIntegerDestroy – *Widget API*

```
void MW_WidgetIntegerDestroy(struct MW_Widget *self);
```

DESCRIPTION
Destroys an integer widget, if self is NULL no operation is performed.

MW_WidgetIntegerInc – *Widget API*

```
void MW_WidgetIntegerInc(struct MW_Widget *self);
```

DESCRIPTION
Increase widget integer value by one.

If resulting value would be greater than maximum and MW_WIDGET_INTEGER_OVERFLOW was passed to MW_WidgetIntegerCreate(3), the value is set to minimum, otherwise it's no-op.

MW_WidgetIntegerLabelGet – *Widget API*

```
const char *MW_WidgetIntegerLabelGet(struct MW_Widget *self);
```

RETURN VALUE
Returns pointer to integer widget label.

MW_WidgetIntegerMaxGet – *Widget API*

```
int32_t MW_WidgetIntegerMaxGet(struct MW_Widget *self);
```

RETURN VALUE
Return widget integer maximum.

MW_WidgetIntegerMaxSet – *Widget API*

```
void MW_WidgetIntegerMaxSet(struct MW_Widget *self, int32_t val);
```

DESCRIPTION
Sets a widget integer maximal value. If current value is greater than new maximum value is set to new maximum.

MW_WidgetIntegerMinGet – *Widget API*

```
int32_t MW_WidgetIntegerMinGet(struct MW_Widget *self);
```

RETURN VALUE
Return widget integer minimum.

MW_WidgetIntegerMinSet – *Widget API*

```
void MW_WidgetIntegerMinSet(struct MW_Widget *self, int32_t val);
```

DESCRIPTION
Sets a widget integer minimum. If value is lesser than new minimum, it's set to new minimum.

MW_WidgetIntegerValGet – *Widget API*

```
int32_t MW_WidgetIntegerValGet(struct MW_Widget *self);
```

RETURN VALUE
Returns current widget integer value.

MW_WidgetIntegerValSet – *Widget API*

```
void MW_WidgetIntegerValSet(struct MW_Widget *self, int32_t val);
```

DESCRIPTION
Sets a widget integer value. Attempts to set value out possible values (i.e. grater than max or lesser than min) are ignored.

B.9 Unsigned Integer Widget

MW_WidgetUIntCreate – *Widget API*

```
enum MW_WidgetUIntFlags {
        /* whether the value could be modified from backend */
        MW_WIDGET_UINT_RDONLY,
        MW_WIDGET_UINT_RDRW,
};

struct MW_Widget *MW_WidgetUIntCreate(const char *id,
                                      uint32_t max, uint32_t val,
                                      uint8_t flags, const char *label);
```

DESCRIPTION
Allocate and initialize an unsigned 32bit integer widget, i.e. an integer value between zero and max.

The id is widget id and must be unique inside window, must not contain colon and must not start with dot.

Initially it's value is set to val.

If flags are set to MW_WIDGET_UINT_RDONLY the value is not allowed to be modified from the backend and MW_WIDGET_UINT_RDRW which allows it to be modified.

The label is string that is optional very short description of the value likely shown in backend somewhere. Could be NULL for no label.

Once the value is changed (in case of MW_WIDGET_UINT_RDRW) by a backend a widget callback, if set, is called.

RETURN VALUE
Function returns pointer to allocated and initialized integer widget or NULL in case of malloc(3) failure.

MW_WidgetUIntDec – *Widget API*

```
void MW_WidgetUIntDec(struct MW_Widget *self, uint32_t dec);
```

DESCRIPTION
Decreases unsigned integer widget value by a given decrement. If the current value is lesser than the decrement the value is truncated (set to zero) and warning via MW_MSG_WARN(3) is printed.

MW_WidgetUIntDestroy – *Widget API*

```
void MW_WidgetIntegerDestroy(struct MW_Widget *self);
```

DESCRIPTION
Destroys an unsigned integer widget, if self is NULL no operation is performed.

MW_WidgetUIntInc – *Widget API*

```
void MW_WidgetUIntInc(struct MW_Widget *self, uint32_t inc);
```

DESCRIPTION
Increases unsigned integer widget value by a given increment. If the current value plus increment is greater than maximum the value is truncated (set to max) and warning via MW_MSG_WARN(3) is printed.

MW_WidgetUIntLabelGet – *Widget API*

```
const char *MW_WidgetUIntLabelGet(struct MW_Widget *self);
```

RETURN VALUE
Returns pointer to the widget label or NULL if case widget didn't have one.

MW_WidgetUIntMaxGet – *Widget API*

```
uint32_t MW_WidgetUIntMaxGet(struct MW_Widget *self);
```

RETURN VALUE
Returns current unsigned integer widget value.

MW_WidgetUIntMaxSet – *Widget API*

```
void MW_WidgetUIntMaxSet(struct MW_Widget *self, uint32_t max);
```

DESCRIPTION

Sets the unsigned integer maximal value. The change is propagated to the backends. If you are about to change both value and maximum use MW_WidgetUIntMaxValSet(3) which sends the change at once and thus possibly avoiding redrawing events in rendering backend.

MW_WidgetUIntMaxValSet – *Widget API*

```
void MW_WidgetUIntMaxValSet(struct MW_Widget *self, uint32_t max, uint32_t val);
```

DESCRIPTION

Sets the unsigned integer value and maximum at once and thus possibly avoiding redrawing events in rendering backend.

MW_WidgetUIntValGet – *Widget API*

```
uint32_t MW_WidgetUIntValGet(struct MW_Widget *self);
```

RETURN VALUE

Returns current unsigned integer widget value.

MW_WidgetUIntValSet – *Widget API*

```
void MW_WidgetUIntValSet(struct MW_Widget *self, uint32_t val);
```

DESCRIPTION

Sets the unsigned integer value. The change is propagated to the backends. If you are about to change both value and maximum use MW_WidgetUIntMaxValSet(3) which sends the change at once and thus possibly avoiding redrawing events in rendering backend.

B.10 Label Widget

MW_WidgetLabelCreate – *Widget API*

```
struct MW_Widget *MW_WidgetLabelCreate(const char *id, uint32_t max_size,
                                       const char *label);
```

DESCRIPTION
Creates a label widget, label is a short one line text.

The id is widget id and must be unique inside window, must not contain colon and must not start with dot.

The maximal number of characters is defined by max_size parameter, if NULL the maximal size is computed from the label parameter.

RETURN VALUE
Function returns pointer to allocated and initialized label widget or NULL in case of malloc(3) failure.

MW_WidgetLabelDestroy – *Widget API*

```
void MW_WidgetLabelDestroy(struct MW_Widget *self);
```

DESCRIPTION
Destroys a label widget, if self is NULL no operation is performed.

MW_WidgetLabelMaxSizeGet – *Widget API*

```
uint32_t MW_WidgetLabelMaxSizeGet(struct MW_Widget *self);
```

RETURN VALUE
Returns maximal widget label text size.

MW_WidgetLabelPrintf – *Widget API*

```
void MW_WidgetLabelPrintf(struct MW_Widget *self, const char *fmt, ...);
```

DESCRIPTION
Printf-like API for setting label widget text. If the resulting string is too long it's truncated.

MW_WidgetLabelTextAppend – *Widget API*

```
void MW_WidgetLabelTextAppend(struct MW_Widget *self, const char *str);
```

DESCRIPTION
Appends a text to the widget label text. If resulting string is too long, it's truncated.

MW_WidgetLabelTextGet – *Widget API*

```
const char *MW_WidgetLabelTextGet(struct MW_Widget *self);
```

RETURN VALUE
Returns widget label text.

MW_WidgetLabelTextSet – *Widget API*

```
void MW_WidgetLabelTextSet(struct MW_Widget *self, const char *label);
```

DESCRIPTION
Sets the widget label text, if text is longer than max_size it's truncated.

B.11 Switch Widget

MW_WidgetSwitchCreate – *Widget API*

```
struct MW_Widget *MW_WidgetSwitchCreate(const char *id, const bool state,
                                        const char *label);
```

DESCRIPTION
Creates a switch widget, that is an widget that is either on or off.

The id is widget id and must be unique inside window, must not contain colon and must not start with dot.

Initially it's state is set to state.

The label is string that is shown in backend somewhere to describe widget's function. Could be NULL for no label.

RETURN VALUE
Function returns pointer to allocated and initialized switch widget or NULL in case of malloc(3) failure.

MW_WidgetSwitchDestroy – *Widget API*

```
void MW_WidgetSwitchDestroy(struct MW_Widget *self);
```

DESCRIPTION
Destroys a switch widget, if self is NULL no operation is performed.

MW_WidgetSwitchLabelGet – *Widget API*

```
const char *MW_WidgetSwitchLabelGet(struct MW_Widget *self);
```

RETURN VALUE
Returns pointer to widget switch label.

MW_WidgetSwitchOff – *Widget API*

```
void MW_WidgetSwitchOff(struct MW_Widget *self);
```

DESCRIPTION
Turns a switch widget off.

MW_WidgetSwitchOn – *Widget API*

```
void MW_WidgetSwitchOn(struct MW_Widget *self);
```

DESCRIPTION
Turns a switch widget on.

MW_WidgetSwitchStateGet – *Widget API*

```
bool MW_WidgetSwitchStateGet(struct MW_Widget *self);
```

RETURN VALUE
Returns current switch state.

MW_WidgetSwitchStateSet – *Widget API*

```
void MW_WidgetSwitchStateSet(struct MW_Widget *self, const bool state);
```

DESCRIPTION
Sets a switch widget state.

B.12 Fd Queue

MW_FdQueue – *Multiplexed I/O*

#include <MW_FdQueue.h>

```
/* Queue itself */
struct MW_FdQueue {
        struct MW_LinkList list;
        int max_fd;
        fd_set set;
};

/* Queue element */
struct MW_Fd {
        struct MW_Fd *next;
        struct MW_Fd *prev;

        int fd;

        uint8_t (*Read)(struct MW_Fd *self, struct MW_FdQueue *queue);

        void *priv;
}
```

DESCRIPTION

MW_FdQueue is abstraction for multiplexed I/O on the top of Unix file descriptor abstraction.

The MW_FdQueue has internally a list of queue members, whose, when MW_FdQueueWait(3) is called are checked for data ready for reading and for each fd with data ready the Read callback is called.

It's possible to add and remove queue elements even from the Read callback. This is especially useful when creating server/client programs.

Beware that by the nature of file descriptor implementation, you must read all data, that are buffered on the file descriptor once read callback is called. Otherwise it's possible that you end up with data ready to be read from kernel, but you are notified about them only when some new data arrives. Also note that even several write(3) calls might be stored in the kernel buffer and send as one, so you get only one notification in such case.

MW_FdQueueCreate – *Multiplexed I/O*

#include <MW_FdQueue.h>

```
struct MW_FdQueue *MW_FdQueueCreate(void);
```

DESCRIPTION
Allocate and initialize struct MW_FdQueue.

RETURN VALUE
Returns pointer to allocated and initialized MW_FdQueue of NULL in case of malloc(3) failure.

MW_FD_INIT – *Multiplexed I/O*

#include <MW_FdQueue.h>

```
MW_FD_INIT(struct MW_Fd *self, int fd,
           uint8_t (*Read)(struct MW_Fd*, struct MW_FdQueue*),
           void *priv)
```

DESCRIPTION
Initialize struct MW_Fd queue element.

MW_FdQueueAdd – *Multiplexed I/O*

#include <MW_FdQueue.h>

```
void MW_FdQueueAdd(struct MW_FdQueue *self, struct MW_Fd *fd);
```

DESCRIPTION
Adds new fd queue member. The member should be initialized by MW_FD_INIT(3).

MW_FdQueueList – *Multiplexed I/O*

#include <MW_FdQueue.h>

```
struct MW_LinkList *MW_FdQueueList(struct MW_FdQueue *self);
```

DESCRIPTION
Returns pointer to double linked list of queue elements (struct MW_Fd).

MW_FdQueueRem – *Multiplexed I/O*

#include <MW_FdQueue.h>

```
void MW_FdQueueRem(struct MW_FdQueue *self, struct MW_Fd *fd);
```

DESCRIPTION
Removes queue element from queue. The element is first unreferenced from queue and then removed from double linked list of queue elements by MW_LinkListRem(3).

MW_FdQueueRemByFd – *Multiplexed I/O*

#include <MW_FdQueue.h>

```
struct MW_Fd *MW_FdQueueRemByFd(struct MW_FdQueue *self, int fd);
```

DESCRIPTION
Removes queue element from queue. When element is found, it's unreferenced from queue and then removed from double linked list of queue elements by MW_LinkListRem(3).

RETURN VALUE
If no queue element with matching file descriptor is found, NULL is returned, otherwise returns pointer to removed element.

MW_FdQueueWait – *Multiplexed I/O*

#include <MW_FdQueue.h>

```
void MW_FdQueueWait(struct MW_FdQueue *self);
```

DESCRIPTION
Wait while data for reading are ready for any queue element.

B.13 Link List

MW_LinkList – *Data Structures*

#include <MW_LinkList.h>

```
struct MW_LinkList {
        void *head; /* pointer to list head */
        void *tail; /* pointer to list tail */

        size_t offset; /* offset to pointer to next element */
        uint32_t type; /* list type */
        uint32_t cnt;  /* number of elements in list */
};
```

DESCRIPTION
The MW_LinkList structure holds all information about list type as well as pointers to first and last list items. The link list type could be either MW_LIST_FIFO or MW_LIST_LIFO and can be optionally combined with MW_LIST_DOUBLE for doubly linked list.

The 'offset' field holds an offset to actual link list pointers, so that you can have one structure in several linked lists. 'cnt' holds actual number of items in list.

The MW_LinkList structure can be dynamically allocated by MW_LinkListCreate(3) or it can be statically initialized by MW_LINK_LIST_INIT(3) or MW_LINK_LIST_PACK(3) macro.

EXAMPLE

```
#include <MW_LinkList.h>
#include <stdio.h>

struct MyListItem {
        int val;
        struct MyListItem *next;
};

int main(void)
{
      /*
       * Initialize list for struct MyListItem with destructor.
       */
        MW_LinkList list = MW_LINK_LIST_PACK(MW_LIST_FIFO, struct MyListItem,
                                             next, free);
        int i;
        struct MyListItem *item;

      /*
       * Fill list with random items.
       */
        for (i = 0; i < 10; i++) {
                item = malloc(sizeof(struct MyListItem));

                if (item == NULL)
                        continue;

                item->val = random() % 20;

                MW_LinkListPush(&list, item);
        }

      /*
       * Print list.
       */
        MW_LINK_LIST_FOREACH(&list, item)
                printf("[%i] ", item->val);

        printf("\n");

      /*
       * Destroy list and free all allocated memory.
       */
        MW_LinkListDestroy(&list);

        return 0;
}
```

MW_LINK_LIST_INIT – *Data Structures*

#include <MW_LinkList.h>

```
MW_LINK_LIST_INIT(struct MW_LinkList *list, type, it_struct, it_next, Free)
```

DESCRIPTION

Macro for initializing struct MW_LinkList.

See MW_LinkListCreate(3) or MW_LinkList(3) for further examples.

EXAMPLE

```
struct MyListItem {
        int val;
        struct MyListItem *next;
};

...

struct MW_LinkList list;

/*
 * Initialize list as a double linked FIFO list with no destructor.
 */
MW_LINK_LIST_INIT(&list, MW_LINK_LIST_FIFO | MW_LINK_LIST_DOUBLE, MyListItem,
                  next, NULL);

...
```

MW_LINK_LIST_PACK – *Data Structures*

#include <MW_LinkList.h>

```
struct MW_LinkList MW_LINK_LIST_PACK(type, it_struct, it_next, Free)
```

DESCRIPTION

Macro for initializing struct MW_LinkList.

See MW_LinkListCreate(3) or MW_LinkList(3) for further examples.

EXAMPLE

```
struct MyListItem {
        int val;
        struct MyListItem *next;
};

...

/*
 * Initialize list as LIFO list using free(3) as destructor.
 */
struct MW_LinkList list = MW_LINK_LIST_PACK(&list, MW_LINK_LIST_LIFO,
                                                MyListItem, next, free);
...
```


#include <MW_LinkList.h>

```
struct MW_LinkList *MW_LinkListCreate(uint32_t type, size_t offset,
                                      void (*Free)(void *item));
```

DESCRIPTION
Allocate and initialize struct MW_LinkList, type could be one of MW_LIST_FIFO or MW_LIST_LIFO optionally combined (by bitwise OR) with MW_LIST_DOUBLE for a doubly linked list.

Offset is offset in list items to next pointer, see offsetof(3) for further information.

Free is pointer to destructor function that is called on every item when destroying list or deleting items.

See MW_LinkList(3) for example.

RETURN VALUE
Function returns pointer to newly allocated and initialized struct MW_LinkList or in case that allocation fails NULL.

MW_LinkListDestroy – *Data Structures*

#include <MW_LinkList.h>

```
void MW_LinkListDestroy(struct MW_LinkList *self);
```

DESCRIPTION
Destroys the linked list.

If the MW_LinkList structure was allocated with MW_LinkListCreate(), it is freed; if a destructor function was defined, it's called on each list item.

If the MW_LinkList structure was statically initialized, no attempt is made to free it; only the destructor function is called for each list item. If no destructor function was specified, the call is effectively a no-op.

MW_LinkListDiscard – *Data Structures*

#include <MW_LinkList.h>

```
void MW_LinkListDiscard(struct MW_LinkList *self, unsigned int cnt);
```

DESCRIPTION
Discards cnt elements from list by doing MW_LinkListPop(3). If there is list element destructor set, calls destructor to every discarded element.

MW_LinkListPop – *Data Structures*

#include <MW_LinkList.h>

```
void *MW_LinkListPop(struct MW_LinkList *self);
```

DESCRIPTION
Pops (removes) an element from the Link List. The element is not freed, just unlinked from the list.

RETURN VALUE
Returns pointer to the removed element or NULL if List is empty.

MW_LinkListPush – *Data Structures*

#include <MW_LinkList.h>

```
void *MW_LinkListPush(struct MW_LinkList *self, void *elem);
```

DESCRIPTION
Pushes an element into the Link List.

RETURN VALUE
Returns pointer to the inserted element, i.e. the same pointer pased as element.

MW_LinkListPushRev – *Data Structures*

#include <MW_LinkList.h>

```
void *MW_LinkListPushRev(struct MW_LinkList *self, void *elem);
```

DESCRIPTION
Does the same as MW_LinkListPush(3), but the element is pushed in reverse manner. So with MW_LINK_LIST_FIFO it's pushed as if the list was MW_LINK_LIST_LIFO and as reverse.

RETURN VALUE
Returns pointer to the inserted element, i.e. the same pointer pased as element.

MW_LINK_LIST_NEXT – *Data Structures*

#include <MW_LinkList.h>

```
item *MW_LINK_LIST_NEXT(MW_LinkList *list, item*)
```

DESCRIPTION
Macro that returns a pointer to the next item in the list.

RETURN VALUE
Pointer to the next item in the list, or NULL in case the item was last in the list.

MW_LINK_LIST_PREV – *Data Structures*

#include <MW_LinkList.h>

```
item *MW_LINK_LIST_PREV(MW_LinkList *list, item*)
```

DESCRIPTION
Macro that returns pointer to previous item in list.

WARNING: this works only on double linked lists and returns random data on single linked lists.

RETURN VALUE
Pointer to next previous item or NULL in case the item was first in list.

MW_LinkListRem – *Data Structures*

#include <MW_LinkList.h>

```
void MW_LinkListRem(struct MW_LinkList *self, void *elem);
```

DESCRIPTION

Removes element from the link list. Note that this operation is O(n) for single linked list as the preceding element must be found. It's however O(1) for double linked list as finding preceding element in double linked list is also O(1).

MW_LINK_LIST_FOREACH – *Data Structures*

#include <MW_LinkList.h>

```
MW_LINK_LIST_FOREACH(list, item)
        code_block;
```

DESCRIPTION
Macro that generates code that iterates on all list members.

EXAMPLE

#include <MW_LinkList.h>

```
struct MyListItem {
        int val;
        struct MyListItem *next;
};

/*
 * Following two functions are equivalent.
 */

struct MyListItem *search1(struct MW_LinkList *list, int val)
{
        struct MyListItem *i;

        for (i = list->head; i != NULL; i = i->next)
                if (i->val == val)
                        return i;

        return NULL;
}


struct MyListItem *search2(struct MW_LinkList *list, int val)
{
        struct MyListItem *i;

        MW_LINK_LIST_FOREACH(list, i)
                if (i->val == val)
                        return i;

        return NULL;
}
```

MW_LinkListSort – *Data Structures*

#include <MW_LinkList.h>

```
void MW_LinkListSort(struct MW_LinkList *self, int (*cmp)(void *, void *));
```

DESCRIPTION

Sorts the linked list using the specified comparison function for comparing elements.

EXAMPLE

```
struct MyListItem {
        int val;
        struct MyListItem *next;
}

/*
 * Returns nonzero if a and b should be swapped.
 */
int compare(struct MyListItem *a, struct MyListItem *b)
{
        return a->val < b->val;
}

/*
 * Function that sorts linked list of struct MyListItem
 */
void sort(struct MW_LinkList *list)
{
        MW_LinkListSort(list, (void*)compare);
}
```
